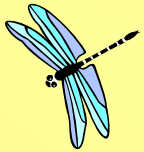
Black Box Software Testing

Spring 2005

Part 3 -- BUG ADVOCACY

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology
and
James Bach
Principal, Satisfice Inc.

These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.



Bug Advocacy

How to

Win Friends,

influence programmers,

and

STOMP BUGs.

ASSIGNED READING: Kaner, Bach, Pettichord, *Bug Advocacy*

Let's do some Bug Advocacy

A bug report is a tool that you use to sell the programmer on the idea of spending her time and energy to fix a bug.

Bug Advocacy?

Bug reports are your primary work product.

- This is what people outside of the testing group will most notice and most remember of your work.

The best tester isn't the one who finds the most bugs or who embarrasses the most programmers. The best tester is the one who gets the most bugs fixed.

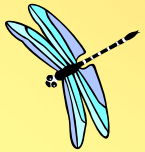
Oops, we need some fine print: When I say "the best tester is the one who gets the most bugs fixed," I am not encouraging bug counting metrics, which are almost always counterproductive. Instead, what I am suggesting is that the effective tester looks to the effect of the bug report, and tries to write it in a way that gives each bug its best chance of being fixed. Also, a bug report is successful if it enables an informed business decision. Sometimes, the best decision is to not fix the bug. The excellent bug report raises the issue and provides sufficient data for a good decision.

Summary of a Bug Workflow

- You find a bug, investigate it, and report it.
- A programmer looks into it and fixes it, decides that a fix will take so long it needs management approval, recommends that it be deferred (fix it later), or determines (argues) it is not a bug.
- The project manager prioritizes the unfixed bugs and may reassign them.
- The project team (with representatives of the key stakeholder groups) reviews deferred bugs and may reprioritize unfixed bugs.
- The test group retests bugs marked as fixed, deferred, or irreproducible and closes them, or adds new information to them and ask that they be reworked or reconsidered. It turns some of these into appeals to the project team.
- (This is a simplified description. Workflows vary significantly.)

So tell me,
What's a Bug?

What is a Bug?



Common Answers

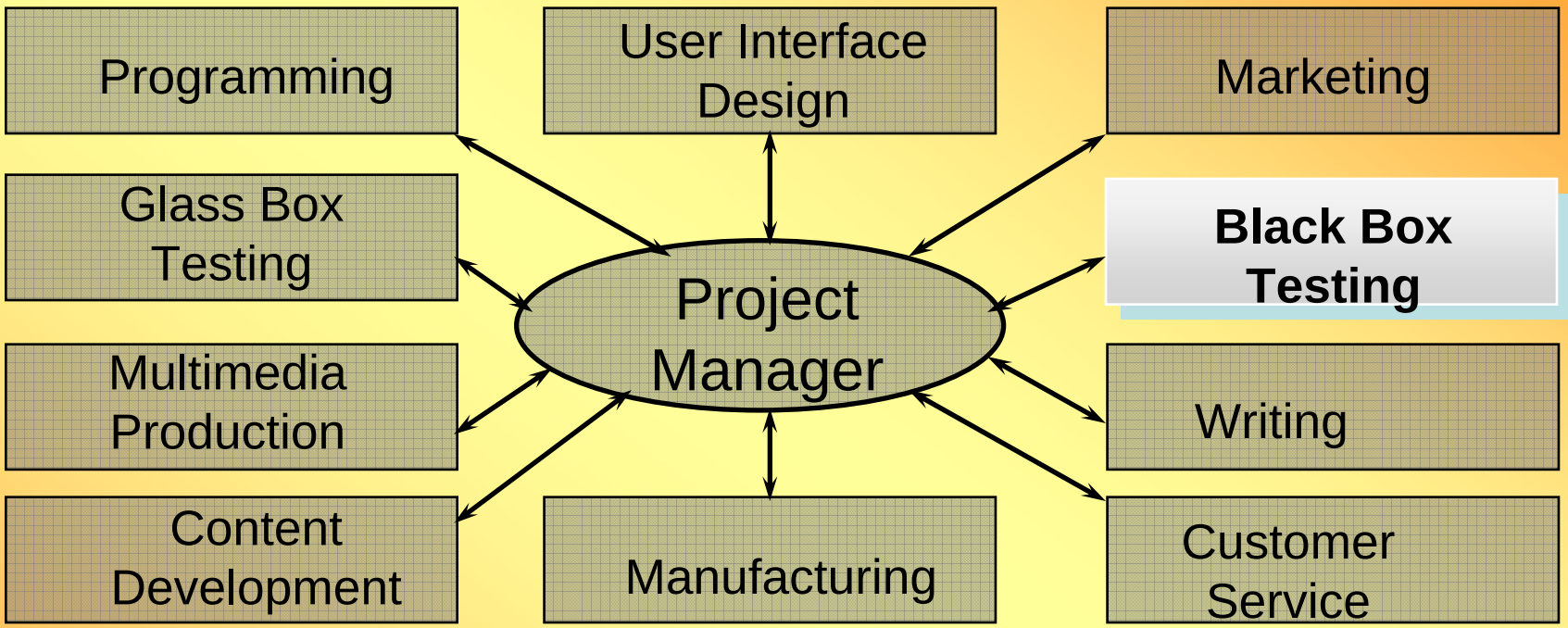
- Deviates from specification
- Deviates from written requirements
- Deviates from documentation
- Coding error (fails to do what the *programmer* intended).
- Fails to meet its design objectives
- Deviates from industry standards
- Deviates from company standards
- Would embarrass the company
- Makes the product less salable
- Generates incorrect results
- Generates confusing results
- Wastes the time of a user
- Interacts badly with other programs or other components of the system
- Interferes with the development, testability or revision of the product.

- Underlying error that causes a failure
- A failure (misbehavior)
- Anything that, if reported, would probably lead to a code change
- Fails to meet reasonable expectations of a user. (Myers)
- Fails to meet reasonable expectations of a stakeholder.
- The term, "software error" includes anything that causes an unnecessary or unreasonable reduction of quality of a software product. ***(Cross-reference to the various definitions of quality).***

What is Quality?

- Here are some traditional definitions:
  - Fitness for use (Dr. Joseph M. Juran)
  - The totality of features and characteristics of a product that bear on its ability to satisfy a given need (ASQ)
  - Conformance with requirements (Philip Crosby)
  - The total composite product and service characteristics of marketing, engineering, manufacturing and maintenance through which the product and service in use will meet expectations of the customer (Armand V. Feigenbaum)
- Note the absence of “conforms to specifications.”
- Quality involves a combination of attributes, such as reliability, usability, maintainability, testability, salability, merchantability, functionality / capability, speed of operation, documentability (it is possible to create good documentation of the product at a non-exorbitant price), localizability, trainability (is the software well enough designed that a trainer can successfully train people in it), etc.

Quality is Multidimensional



When you sit in a project team meeting, discussing a bug, a new feature, or some other issue in the project, you must understand that each person in the room has a different vision of what a “quality” product would be. Fixing bugs is just one issue.
The next slide gives some examples.

Quality is Multidimensional: Different People, Different Visions

- **Localization Manager:** A good product is easy to translate and to modify to make it suitable for another country and culture. Few experienced localization managers would consider acceptable a product that must be recompiled or relinked to be localized.
- **Tech Writers:** A high quality program is easily explainable. Aspects of the design that are confusing, unnecessarily inconsistent, or hard to describe are marks of bad quality.
- **Marketing:** Customer satisfiers drive people to buy the product and tell their friends about it. A Marketing Manager trying to add new features to the product generally believes he's trying to improve the product.
- **Customer Service:** Good products are supportable. They have been designed to help people solve their own problems or to get help quickly.
- **Programmers:** Great code is maintainable, well documented, easy to understand, well organized, fast and compact.

What is Quality?

- Joseph Juran distinguishes between *Customer Satisfiers* and *Dissatisfiers* as key dimensions of quality:
- *Customer Satisfiers*
  - the right features
  - adequate instruction
- *Dissatisfiers*
  - unreliable
  - hard to use
  - too slow
  - incompatible with the customer's equipment

Quality:
What Should We Report?

- I like Gerald Weinberg's definition:

  Quality is value to some person

- But consider the implication:
  - It's appropriate to report any deviation from high quality as a software error.
  - Therefore many issues will be reported that will be errors to some and non-errors to others.

Software Errors: What Kind of Error?

- You'll report all these types of problems, but it's valuable to keep straight in your mind, and on the bug report, which type you're reporting.
  - **Coding Error:** The program doesn't do what the programmer would expect it to do.
  - **Design Issue:** It's doing what the programmer intended, but a reasonable customer would be confused or unhappy with it.
  - **Requirements Issue:** The program is well designed and well implemented, but it won't meet one of the customer's requirements.
  - **Documentation / Code Mismatch:** Report this to the programmer (via a bug report) and to the writer (usually via a memo or a comment on the manuscript).
  - **Specification / Code Mismatch:** Sometimes the spec is right; sometimes the code is right and the spec should be changed.

Selling Bugs

- **Time is in short supply. If you want to convince the programmer to spend time fixing your bug, you may have to sell her on it.**

  (Your bug? How can it be your bug? The programmer made it, not you, right? It's the programmer's bug. Well, yes, but you found it so now it's yours *too.)*

- **Sales revolves around two fundamental objectives:**
  - **Motivate the buyer (*Make her WANT to fix the bug.)***
  - **Overcome objections (*Get past her excuses and reasons for not fixing the bug.)***

Motivating the Bug Fixer

- Some things often make programmers want to fix the bug:
  - It looks really bad.
  - It looks like an interesting puzzle and piques the programmer's curiosity.
  - It will affect lots of people.
  - Getting to it is trivially easy.
  - It has embarrassed the company, or a bug like it embarrassed a competitor.
  - One of its cousins embarrassed the company or a competitor.
  - Management (that is, someone with influence) has said that they really want it fixed.
  - You've said that you want the bug fixed, and the programmer likes you, trusts your judgment, is susceptible to flattery from you, owes you a favor or accepted bribes from you.

Overcoming Objections

- These make programmers resist spending time on a bug:
  - The programmer can't replicate the defect.
  - Strange and complex set of steps required to induce the failure.
  - Not enough information to know what steps are required, and it will take a lot of work to figure them out.
  - The programmer doesn't understand the report.
  - Unrealistic (e.g. "corner case")
  - It will take a lot of work to fix the defect.
  - A fix will introduce too much risk into the code.
  - No perceived customer impact
  - Unimportant (no one will care: minor error or unused feature.)
  - Management doesn't care about bugs like this.
  - That's not a bug, it's a feature.
  - The programmer doesn't like / trust you (or the customer who is complaining about the bug).

Motivating bug fixes through better research of failure conditions

Motivating The Bug Fix: Looking At The Failure

- Some vocabulary
  - An ***error*** (or ***fault***) is a design flaw or a deviation from a desired or intended state.
  - An error won't yield a failure without the ***conditions*** that trigger it. Example, if the program yields 2+2=5 on the 10th time you use it, you won't see the error before or after the 10th use.
  - The ***failure*** is the program's actual incorrect or missing behavior under the error-triggering conditions.
  - A ***symptom*** might be a characteristic of a failure that helps you recognize that the program has failed.
  - ***Defect*** might refer to the failure or to the underlying error.
    - Nancy Leveson (Safeware) draws useful distinctions between errors, hazards, conditions, and failures.

Motivating The Bug Fix: Looking At The Failure

VOCABULARY EXAMPLE

- Here's a defective program
  - INPUT A
  - INPUT B
  - PRINT A/B
- What is the fault?
- What is the critical condition?
- What will we see as the failure?

Motivating the Bug Fix

- When you run a test and find a failure, you're looking at a symptom, not at the underlying fault. You may or may not have found the best example of a failure that can be caused by the underlying fault.
- Therefore you should do some follow-up work to try to prove that a defect:
  - **is more serious than it first appears.**
  - **is more general than it first appears.**

Motivating the Bug Fix: Make it More Serious

LOOK FOR FOLLOW-UP ERRORS

- When you find a coding error, the program is in a state that the programmer did not intend and probably did not expect. There might also be data with supposedly impossible values.
- The program is now in a vulnerable state. Keep testing it and you might find that the real impact of the underlying fault is a much worse failure, such as a system crash or corrupted data.
- I do three types of follow-up testing:
  - ***Vary my behavior*** (change conditions by changing what I do)
  - ***Vary the options and settings of the program*** (change conditions by changing something about the program under test).
  - ***Vary the*** software and hardware ***environment***.

Follow-Up: Vary Your Behavior

- Keep using the program after you see the problem.
- **Bring it to the failure case again (and again).** If the program fails when you do X, then do X many times. Is there a cumulative impact?
- Try things that are ***related to the task*** that failed. For example, if the program unexpectedly but slightly scrolls the display when you add two numbers, try tests that affect adding or that affect the numbers. Do X, see the scroll. Do Y then do X, see the scroll. Do Z, then do X, see the scroll, etc. (If the scrolling gets worse or better in one of these tests, follow that up, you're getting useful information for debugging.)
- Try things that are ***related to the failure***. If the failure is unexpected scrolling after adding, try scrolling first, then adding. Try repainting the screen, then adding. Try resizing the display of the numbers, then adding.
- Try entering the numbers more quickly or ***changing the speed*** of your activity in some other way.
- And ***try the usual exploratory testing techniques*****.** Example: try some interference tests. Stop or pause the program or swap it just as the program is failing. Or try it while the program is doing a background save. Does that cause data loss corruption along with this failure?

Follow-Up: Vary Options and Settings

- In this case, the steps to achieve the failure are taken as given. Try to reproduce the bug with the program in a different state:
  - Use a different database.
  - Change the values of persistent variables.
  - Change how the program uses memory.
  - Change anything that looks like it might be relevant that allows you to change as an option.
- For example, suppose the program scrolls unexpectedly when you add two numbers. Maybe you can change the size of the program window, or the precision (or displayed number of digits) of the numbers, or background the activity of the spell checker.

Follow-Up: Vary the Configuration

- A bug might show a more serious failure if you run replicate with less memory, a different printer, more device interrupts coming in etc.
  - If it might involve timing, use a really slow (or fast) computer or peripheral.
  - If there is a video problem, try other resolutions on the video card. Try displaying MUCH more (less) complex images.
- Note that we are not:
  - checking standard configurations
  - asking how broad the circumstances that produces the bug.
- What we're asking is whether there is a particular configuration that will show the bug more spectacularly.
- Returning to the example (unexpected scrolling when you add two numbers), try things like:
  - Different video resolutions
  - Different settings on a wheel mouse that does semi-automated scrolling
  - An NTSC (television) signal output instead of a traditional (XGA or SVGA, etc.) monitor output.

Follow-up: Bug New to This Version?

- In many projects, an old bug (from a previous shipping release of the program) might not be taken very seriously if there weren't lots of customer complaints.
  - (If you know it's an old bug, check its complaint history.)
  - The bug will be taken more seriously if it is new.
  - You can argue that it should be treated as new if you can find a new variation or a new symptom that didn't exist in the previous release. What you are showing is that the new version's code interacts with this error in new ways. That's a new problem.
  - This type of follow-up testing is especially important during a maintenance release that is just getting rid of a few bugs. Bugs won't be fixed unless they were (a) scheduled to be fixed because they are critical or (b) new side effects of the new bug fixing code.

Motivating the Bug Fix: Show it is More General

LOOK FOR CONFIGURATION DEPENDENCE

- Bugs that don't fail on the programmer's machine are much less credible (to that programmer). If they are configuration dependent, the report will be much more credible if it identifies the configuration dependence directly (and so the programmer starts out with the expectation that it won't fail on all machines.)

Question: **How many programmers does it take to change a light bulb?**

Answer: ***What's the problem? The bulb at my desk works fine!***

Look for Configuration Dependence

- In the ideal case (standard in many companies), test on 2 machines
  - Do your main testing on Machine 1. Maybe this is your powerhouse: latest processor, newest updates to the operating system, fancy printer, video card, USB devices, huge hard disk, lots of RAM, cable modem, etc.
  - When you find a defect, use Machine 1 as your bug reporting machine and replicate on Machine 2. Machine 2 is totally different. Different processor, different keyboard and keyboard driver, different video, barely enough RAM, slow, small hard drive, dial-up connection with a link that makes turtles look fast.
  - Some people do their main testing on the turtle and use the power machine for replication.
  - Write the steps, one by one, on the bug form at Machine 1. As you write them, try them on Machine 2. If you get the same failure, you've checked your bug report while you wrote it. (A valuable thing to do.)
  - If you don't get the same failure, you have a configuration dependent bug. Time to do troubleshooting. But at least you know that you have to.
- **AS A MATTER OF GENERAL GOOD PRACTICE, IT PAYS TO REPLICATE EVERY BUG ON A SECOND MACHINE.**

Motivating the Bug Fix: Show it is More General

UNCORNER YOUR CORNER CASES

- We test at extreme values because these are the most likely places to show a defect. *But once we find the defect, we don't have to stick with extreme value tests.*
- Try mainstream values.
  - These are easy settings that should pose no problem to the program.
  - If you replicate the bug, write it up, referring primarily to these mainstream settings. This will be a very credible bug report.

Motivating the Bug Fix: Show it is More General

UNCORNER YOUR CORNER CASES

- If mainstream values don't yield failure, but the extremes do, troubleshoot around the extremes.
  - Is the bug tied to a single setting (a true corner case)?
  - Or is there a small range of cases? What is it?
  - In your report, identify the narrow range that yields failures. The range might be so narrow that the bug gets deferred. That might be the right decision. In some companies, the product has several hundred open bugs a few weeks before shipping. They have to decide which 300 to fix (the rest will be deferred). Your reports help the company choose the right 300 bugs to fix, and help people size the risks associated with the remaining ones.

Why are there errors?

Why are there Errors?

- New testers often conclude that programmers on their project are incompetent or unprofessional.
  - This is counterproductive. It leads to infighting instead of communication, and it leads to squabbling over bugs instead of research and bug fixing.
  - Programmers find and fix almost all of their own bugs.
  - Bugs come into the code for many reasons. It's worth considering some common systematic (as distinct from poor individual performance) factors. You will learn to vary your strategic approaches as you learn your companies' weaknesses.

Why are there Errors?

- Bugs come into the code for many reasons:
  - The major cause of error is that programmers deal with tasks that aren't fully understood or fully defined. This is said in many different ways. For example:
    - Tom Gilb and Dick Bender quote industry-summary statistics that 80% of the errors, or 80% of the effort required to fix the errors, are caused by bad requirements;
    - Roger Sherman recently summarized research at Microsoft that the most common underlying issue in bug reports involved a need for new code.

    If you graduated from a Computer Science program, how much training did you have in task analysis? Requirements definition? Usability analysis? Negotiation and clear communication of negotiated agreements? Not much? Hmmmm . . . .

Why are there Errors?

- Some companies don't give their programmers enough time to design, bulletproof, or test their code. Another Sherman quote: "Bad schedules are responsible for most quality problems."
- Late design changes result in last minute code changes, which are likely to have errors.
- Some third-party components introduce bugs. Your program might rely on a large suite of small components that display a specific type of object, filter data in a special way, drive a specific printer, etc. Many of these tools, bought from tool vendors or hardware vendors, are surprisingly buggy. Others work, but they aren't fully compatible with common test automation tools.
- Failure to use source control tools creates characteristic bugs. For example, if a bug goes away, comes back, goes away, comes back, goes away, comes back, then ask how the programming staff makes sure it's linking the most recent version of each module when it builds a version for you to test.

Why are there Errors?

- Some programs or tasks are inherently complex. Boris Beizer talks perceptively about the locality problem in software. Think about a coding error and the symptoms it causes. There's no assurance that symptoms will be close in time, space, or severity to the underlying bug. They may appear later, or when working with a different part of the program, and may seem much more or less serious than the coding error.
- Some programmers (some platforms) work with poor tools. Weak compilers, style checkers, debuggers, profilers, etc. make it too easy to get bugs or too hard to find bugs.
- Similarly, some third party hardware, or its drivers, are non-standard and don't respond properly to standard system calls. Incompatibility with hardware is often cited as the largest single source of customer complaints into technical support groups.
- When one programmer tries to modify another programmer's code, there's lots of room for miscommunication and error.
- And, sometimes people just make mistakes.

Summing Up

- Bug reporting is persuasive writing
- Your credibility as a reporter is essential to get reports taken seriously.
- Keeping reports honest and unexaggerated, but focus the report around the most serious failure, not the first failure you saw. Remember: a failure is a misbehavior caused by an underlying error. An error can cause many different failures.
- If failure varies as you change conditions, write the main description around the most serious failure, then describe other conditions at the end of the report as troubleshooting notes.
- We can do common types of follow-up testing to expose more serious problems. Some bugs need no follow-up testing. For the others, spend perhaps 15 minutes but give yourself more time if you think it will pay off.
- There is the question of what to report. I report anything that any stakeholder with influence might want fixed (changed). This is based on Weinberg's definition of quality.